Getting started with

Habari Client for OpenMQ

Version 2.9

Trademarks

Habari is a registered trademark of Michael Justin and is protected by the laws of Germany and other countries. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Embarcadero, the Embarcadero Technologies logos and all other Embarcadero Technologies product or service names are trademarks, service marks, and/or registered trademarks of Embarcadero Technologies, Inc. and are protected by the laws of the United States and other countries. Microsoft, Windows, Windows NT, and/or other Microsoft products referenced herein are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other brands and their products are trademarks of their respective holders.

Contents

What's new in version 2.9?....6
Fixed....6
Improvements....6

Broker and library limitations....7
Feature Matrix....7
Transactional Ack....7
STOMP 1.1 and STOMP 1.2 support....7
Queue browser....7
Message types....7

Quick start guide for OpenMQ....8
Download and Installation....8
Start the Broker....8
Configuration....8
Test the Stomp connection....9
GlassFish v4 configuration....9

Installation....11
Requirements....11
Development Environment....11
TCP/IP Communication Libraries....11
Supported libraries....11

Communication Adapters....12
Introduction....12
Configuration of communication adapters....12
Registration of communication adapter class....12
Available communication adapters....13
Which communication adapter class should I use?....13
Limitations of the Synapse communication adapter class....13
SSL communication adapter classes....13

The Programming Model....15

Tutorials....16
Quick Start Tutorial....16
Setting up the project....16
Adding code to the project....16
Run the demo....18
Check for memory leaks....18
Tutorial source code....18
Map Message Tutorial....18
Setting up the project....18
Adding code to the project....19
Run the demo....21
Map Message Conversion with Apache ActiveMQ....21
Tutorial source code....21

Connections and Sessions....22
Step-by-Step Example....22
Overview....22
Add required units....22
Creating a new Connection....22
Connection URL Parameters....23
Creating a Session....23
Using the Session....23
Closing a Connection....23
Session types overview....24
Connection URL parameters....25
Message Receipts....25
Subscription Receipts....25
Disconnect Receipts....25
TCP Keep-Alive (only supported with Indy)....26
Heart-beating Support....26
Transacted Sessions....27
Create a transacted session....27
Send messages....27
Committing a transaction....27
Rolling back a transaction....28
Transacted message acknowledgement....28
Failover Support....29
Failover Transport Options....29
Pooled Connection Factory....30

Destinations....32
Introduction....32
Create a new Destination....32
Queues....32
Topics....33

Producer and Consumer....34
Message Producer....34
Broker-specific note....34
Message Consumer....34
Synchronous....35
Asynchronous....35
Synchronous Receive....35
Should I use synchronous receive or IMessageListener?....36

Durable Subscriptions....37
Description....37
Creation....37
Examples....37

Temporary Queues....38
Introduction....38
Library Support....38
Resource Management....38

Message Options....39
JMS Standard Properties....39
JMS properties for outgoing messages....39
JMS properties for incoming messages....39
Reserved Names....40
JMS Selectors....41
Supported message brokers 41

Object Messages....42
Introduction....42
Object Serialization....42
Message Transformers....42
Memory Management....42
Assign a Message Transformer....43
Create and Send an ObjectMessage....43
Complete Example using NativeXml....43

Stomp 1.2....45
Connection Configuration....45
Specification....46
Sending heart-beat signals....46
Checking for incoming heartbeats....47
Reading server-side heartbeats....47

Example Applications....48
Broker-independent example projects....49
Broker-specific preparation....49
Shared units for demo projects....49
SSL communication adapter units....49
ConsumerTool....50
Examples....51
ProducerTool....52
Examples....52
Performance Test....54
Throughput Test Tool....56
Examples....56

Logging with Log4D....57
Introduction....57
IDE configuration....57
Minimal example....57
Log4D configuration file....57
Example file "log4d.props"....58

Conditional Symbols....60
HABARI_LOGGING....60
HABARI_RAW_TRACE....60

Useful Units....61
BTStreamHelper....61
BTJavaPlatform....61

Library Limitations....62
Session....62
Transacted Sessions....62
MessageConsumer....62
How do I implement synchronous receive from multiple destinations?....62
Message Properties....63
Only string data type supported by Stomp....63
Multi Threading....63
GUI application detection using System.IsConsole....63
Other limitations....63
ISAPI modules....64

Online Resources....65
Third-Party Libraries....65
Specifications....66
Online Articles....67
Online Videos....67

Support....69
Bug reports....69
Support inquiries....69
Advanced support....69

Index....70

What's new in version 2.9?

Fixed

Throughput test	The throughput test application uses the subscribe.receipt=true connection parameter to ensure that the broker is ready before the producer starts sending messages (1108)

Improvements

Connection factory	As a preparation for further code streamlining between library versions, a new API compatible factory class TBTConnectionFactory is included in this version which will replace TBTJMSConnectionFactory in the next major release (1166)
Password logging	If Log4D logging is enabled, the passcode header value will be not be logged but replaced by *** in the log output (1139)
Optional units	Several units which are not covered by basic support have been moved to the source/optional directory (BTMessageTransformer*, BTStreamhelper, SSL support units) (1161, 1162)
Refactorings	Refactorings reduce unit dependencies and hard type casts in several places (1085, 1094)
Cleanup	Removed unused procedure TBTStompClient.Connect(const AURI: IURI) (1165)
Documentation	More documentation improvements and streamlining between library versions

Broker and library limitations

Feature Matrix

The Habari Client library feature matrix gives an overview about supported features for the different broker versions. You can find the current version of the feature matrix at

https://www.habarisoft.com/

Transactional Ack

The OpenMQ STOMP adapter does not support transactional message acknowledgment.

STOMP 1.1 and STOMP 1.2 support

The OpenMQ STOMP adapter does not support STOMP 1.1.

STOMP 1.2 is supported in OpenMQ 5.1 with limitations.

Some limitations which have been found so far:

- heart-beating is not supported
- the broker does not accept standard STOMP 1.1/1.2 header names which contain hyphens, for example “content-type”

other limitations may exist. Some known limitations are also handled in the unit tests, by excluding them from the test suite (see source code).

Queue browser

The OpenMQ STOMP adapter does not support queue browsing.

Message types

It is not possible to detect the message body type for incoming messages (sent from the OpenMQ broker to the Stomp client) because there is no indicator in the message header for the type. As a workaround, Habari Client for OpenMQ uses a non-standard Stomp header to indicate the message body type:

- messagetype=text
- messagetype=byte

This header will be included in outgoing messages (from the Habari Stomp client to the message broker) and helps the receiver to identify the message type.

Quick start guide for OpenMQ

Download and Installation

The OpenMQ 4.5 web page is located at http://mq.java.net/4.5.html.

Note that this installation documentation uses the Windows installer, however the Habari Client for OpenMQ library would work with installations of OpenMQ on other operating system platforms as well.

Installation steps for OpenMQ 4.5:

1. download the binary image file with installer (for Microsoft Windows x86) from the OpenMQ web page
2. unpack the installer
3. navigate to the directory `openmq4_5-installer`
4. start the installer script `installer.vbs`

When the installer application is started, a graphical application will display allow you to install and configure Message Queue.

Start the Broker

You are ready to start the OpenMQ server now via the `mq/bin/imqbrokerd` command, which will run the broker and display log messages in a window

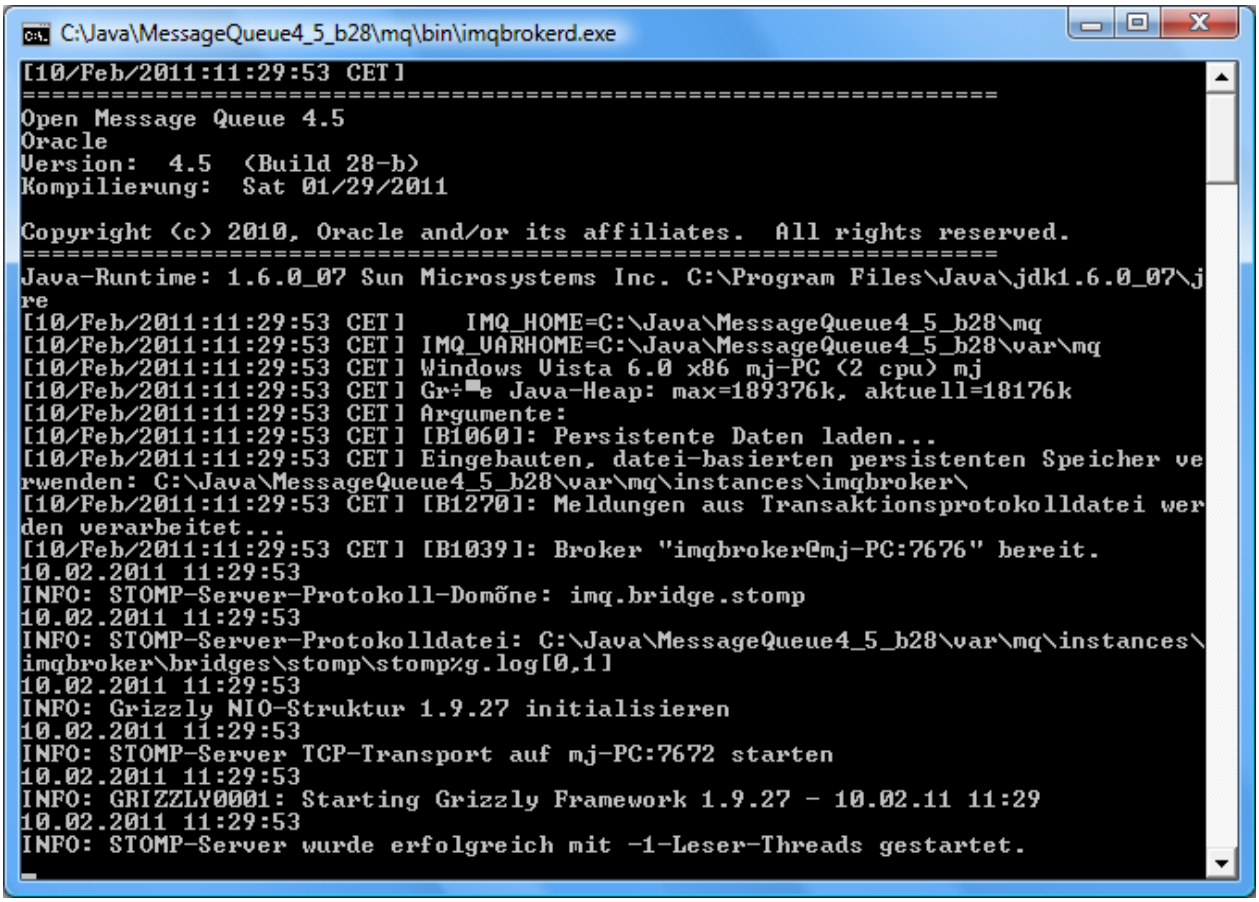
Configuration

Now you will need to enable Stomp support in the broker and configure a user name and password for the broker bridge.

1. stop the broker
2. navigate to the configuration folder `.../instances/<instance name>/props` (this folder will be created when the broker started), for example `C:\MessageQueue-4.5\var\mq\instances\imqbroker\props`
3. open the configuration file `config.properties` with a text editor
4. add the following lines to configure the Stomp adapter with a test admin account:

```
imq.bridge.admin.user=admin
imq.bridge.admin.password=admin
imq.bridge.activelist=stomp
imq.bridge.enabled=true
```

Save the file. You are ready to start the broker now with Stomp support.


C:\Java\MessageQueue4_5_b28\mq\bin\imqbrokerd.exe
[10/Feb/2011:11:29:53 CET]
Open Message Queue 4.5
Oracle
Version: 4.5 (Build 28-b)
Kompilierung: Sat 01/29/2011
Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
Java-Runtime: 1.6.0_07 Sun Microsystems Inc. C:\Program Files\Java\jdk1.6.0_07\jre
[10/Feb/2011:11:29:53 CET] IMQ_HOME=C:\Java\MessageQueue4_5_b28\mq
[10/Feb/2011:11:29:53 CET] IMQ_VARHOME=C:\Java\MessageQueue4_5_b28\var\mq
[10/Feb/2011:11:29:53 CET] Windows Vista 6.0 x86 mj-PC (2 cpu) mj
[10/Feb/2011:11:29:53 CET] Gr÷▪e Java-Heap: max=189376k, aktuell=18176k
[10/Feb/2011:11:29:53 CET] Argumente:
[10/Feb/2011:11:29:53 CET] [B1060]: Persistente Daten laden...
[10/Feb/2011:11:29:53 CET] Eingebauten, datei-basierten persistenten Speicher verwenden: C:\Java\MessageQueue4_5_b28\var\mq\instances\imqbroker\
[10/Feb/2011:11:29:53 CET] [B1270]: Meldungen aus Transaktionsprotokolldatei werden verarbeitet...
[10/Feb/2011:11:29:53 CET] [B1039]: Broker "imqbroker@mj-PC:7676" bereit.
10.02.2011 11:29:53
INFO: STOMP-Server-Protokoll-Domõne: imq.bridge.stomp
10.02.2011 11:29:53
INFO: STOMP-Server-Protokolldatei: C:\Java\MessageQueue4_5_b28\var\mq\instances\imqbroker\bridges\stomp\stomp%g.log[0,1]
10.02.2011 11:29:53
INFO: Grizzly NIO-Struktur 1.9.27 initialisieren
10.02.2011 11:29:53
INFO: STOMP-Server TCP-Transport auf mj-PC:7672 starten
10.02.2011 11:29:53
INFO: GRIZZLY0001: Starting Grizzly Framework 1.9.27 - 10.02.11 11:29
10.02.2011 11:29:53
INFO: STOMP-Server wurde erfolgreich mit -1-Leser-Threads gestartet.

As you can see, the Stomp bridge is running and using TCP port 7672. The Habari Client for OpenMQ library will use port 7672 by default.

Test the Stomp connection

To test the Stomp connection, you can use the ProducerTool demo application in the directory `<Habari>\demo\producertool`.

If you start the ProducerTool without command line parameters, it will send 10 messages to the broker. For a description of the parameters, please see chapter "ProducerTool" (p. 52).

GlassFish v4 configuration

In the default installation of GlassFish v4, OpenMQ is not started automatically when the broker starts, so the message broker configuration file for the default domain "domain1" is not present yet.

Follow these steps to initialize OpenMQ in GlassFish v4:

- start GlassFish

- Trigger the lazy initialisation of the embedded MQ broker - run `<glassfish>/bin/asadmin jms-ping`
- edit the `config.properties` file to activate the STOMP bridge:

```
vim <glassfish>/domains/domain1/imq/instances/imqbroker/props/config.properties
```

- restart GlassFish

You can also initialize OpenMQ through the GlassFish web management interface: create a new JMS resource, for example a topic.

To verify that the broker has started the STOMP bridge, use netstat:

```
netstat -l | grep "7672"
```

Or check the log file of Open Message Queue:

```
cat <glassfish>/domains/domain1/imq/instances/imqbroker/log/log.txt | grep "stomp"
```

The STOMP bridge log file is located in directory

```
cat <glassfish>/domains/domain1/imq/instances/imqbroker/bridges/stomp
```

And after a successful initialization the STOMP bridge log file should contain these lines:

```
Feb 12, 2014 12:40:48 PM
INFO: STOMP server log domain: imq.bridge.stomp
Feb 12, 2014 12:40:48 PM
INFO: STOMP server log file:
.../domains/domain1/imq/instances/imqbroker/bridges/stomp/stomp%g.log[0,1]
Feb 12, 2014 12:40:48 PM
INFO: Initialize Grizzly NIO framework 2.3.1
Feb 12, 2014 12:40:48 PM
INFO: Start STOMP server TCP transport on example.com.de:7672
```

Installation

Requirements

Development Environment

- **Embarcadero Delphi** 2009 Update 4 or higher
- **Free Pascal** 2.6.4 or higher

TCP/IP Communication Libraries

Supported libraries

Internet Direct (Indy) 10.6

The communication adapter for Indy supports both GUI-based and console mode applications, and works with Delphi 2009 and higher and Free Pascal 2.6.0 or higher.

Synapse Release 40

The communication adapter for Synapse supports both GUI-based and console mode applications, and works with Delphi 2009 and higher and Free Pascal 2.6.0 or higher.

Communication Adapters

Introduction

Habari Client for OpenMQ uses communication adapters as an abstraction layer for the TCP/IP library. All connections create their own internal instance of the adapter class.

Configuration of communication adapters

No configuration is required for the communication adapters. Applications specify communication and connection options in URL parameters or connection class properties or connection factory settings.

Registration of communication adapter class

A communication adapter implementation can be prepared for usage by simply adding its Delphi unit to the project. Example:

```
program ClientUsingIndy;

uses
  BTCommAdapterIndy, // use Internet Direct (Indy)
  BTConnectionFactory, BTJMSInterfaces,
  SysUtils;
...
```

Behind the scenes, the communication adapter class will register itself with the communication adapter manager in the BTAdapterRegistry unit.

Default adapter class

Applications typically use only one of the available communication adapter classes for all connections.

The library allows to register two or more adapter classes and switch at run-time, using methods in the adapter registry in unit BTAdapterRegistry - this feature is mainly for tests and demonstration purposes.

If more than one communication adapter is in the project, the **first** adapter class in the list will be the default adapter class. Example:

```
program ClientUsingIndyOrSynapse;

uses
  BTCommAdapterIndy, // use Internet Direct (Indy) as default adapter class
  BTCommAdapterSynapse, // and register the Synapse adapter class
  BTConnectionFactory, BTJMSInterfaces,
  SysUtils;
```

```
...
```

The default adapter class can be changed at run-time by setting the adapter class either by its name or by its class type.

Available communication adapters

The library includes two adapter classes for TCP/IP libraries, one for Indy (Internet Direct) and one for Synapse.

Adapter Class	**Unit**
TBTCommAdapterIndy	BTCommAdapterIndy
TBTCommAdapterSynapse	BTCommAdapterSynapse

Table 1: Communication Adapters

Which communication adapter class should I use?

The Internet Direct (Indy) communication adapter class is recommended.

Limitations of the Synapse communication adapter class

- The Synapse library does not support the ConnectTimeout property in synchronous socket operation mode, as connect timeouts are handled by the operating system. Indy uses a background thread to abort the connect operation.[1]
- Release 40 of Ararat Synapse is used for Habari Client library development and tests. This is the last announced release, dated April 24, 2012. If you use a newer release of Ararat Synapse, please let me know if you encounter any API incompatibilities or other problems.

SSL communication adapter classes

The library also includes two **experimental** adapter classes for usage with OpenSSL, one for Indy (Internet Direct) and one for Synapse. Support for these adapter classes is not included in the basic support package. The units for these classes are in the source\optional folder. A demo application is included in common-producertool-ssl.

1 http://www.ararat.cz/synapse/doku.php/public:howto:connecttimeout

Adapter Class	Unit
TBTCommAdapterIndySSL	BTCommAdapterIndySSL
TBTCommAdapterSynapseSSL	BTCommAdapterSynapseSSL

Table 2: Communication Adapters with SSL Support

The Programming Model

Habari Client libraries use a programming model which is based on message producers and message consumers, sessions, connections and connection factories.

The basic API is the same for all library versions to allow easy migration between supported message brokers (with the exception of broker-specific features).

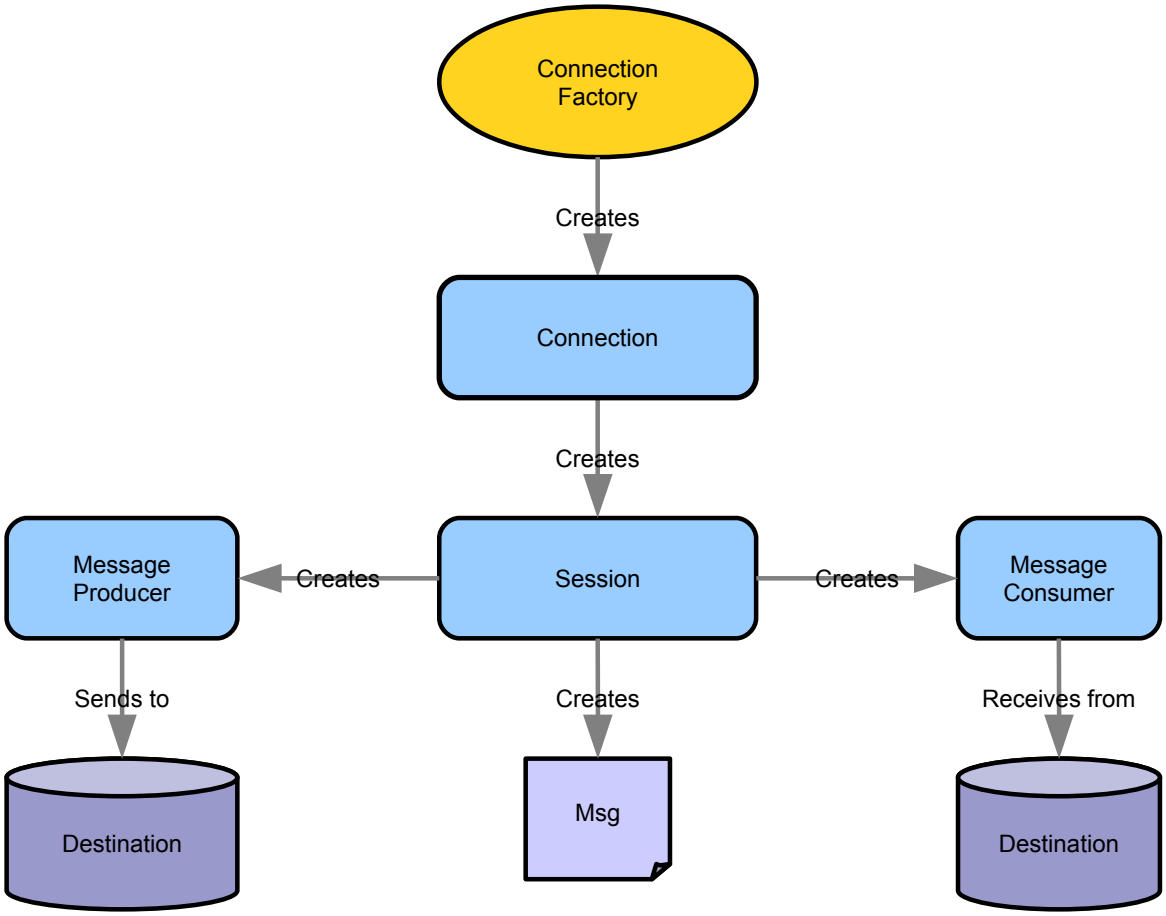


Illustration 1: Programming Model

Tutorials

Quick Start Tutorial

This tutorial provides a very simple and quick introduction to Habari Client for OpenMQ by walking you through the creation of a simple "Hello World" application. Once you are done with this tutorial, you will have a general knowledge of how to create and run Habari applications.

This tutorial takes less than 10 minutes to complete.

Setting up the project

To create a new project:

1. Start the Delphi IDE.
2. In the IDE, choose File > New > VCL Forms Application – Delphi
3. Choose Project > Options … to open the Project Options dialog
4. In the options tree on the left, select 'Delphi Compiler'
5. Add the source directory of Habari and the Synapse source directory to the 'Search path'
6. Choose Ok to close the Project Options dialog
7. Save the project as HelloMQ

Now the project is created and saved.

You should see the main form in the GUI designer now.

Adding code to the project

To use the Habari Client for OpenMQ library, you need to add the required units to the source code.

8. Switch to Code view (F12)
9. Add the required units to the interface uses list:

```
uses
  BTConnectionFactory,
  BTJMSInterfaces,
  BTCommAdapterSynapse,
  // auto-generated unit references
  Windows, Messages, SysUtils, ...
```

10. Compile and save the project.
11. Switch to Design view (F12), go to the Tool palette (Ctrl+Alt+P) and select TButton, add a Button to the form.
12. Double click on the new button to jump to the Button Click handler
13. Add the following code to send the message:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Factory := TBTConnectionFactory.Create;
  Connection := Factory.CreateConnection;
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Producer := Session.CreateProducer(Destination);
  Producer.Send(Session.CreateTextMessage('Hello world!'));
  Connection.Close;
end;
```

14. Add a second button and double click on the new button to jump to the Button Click handler
15. Add the following code to receive and display the message:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Consumer: IMessageConsumer;
  Msg: ITextMessage;
begin
  Factory := TBTConnectionFactory.Create;
  Connection := Factory.CreateConnection;
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Consumer := Session.CreateConsumer(Destination);
  Msg := Consumer.Receive(1000) as ITextMessage;
  if Assigned(Msg) then
    ShowMessage(Msg.Text);
  Connection.Close;
end;
```

16. Compile and save the project

Run the demo

- Launch the message broker
- Start the application
- Click on Button 1 to send the message to the queue
- Click on Button 2 to receive the message and display it

You can run two instances of the application at the same time, and also on different computers if the IP address of the message broker is used instead of localhost.

Check for memory leaks

To verify that the program does not cause memory leaks, insert a line in the project file HelloMQ.dpr:

```
program HelloMQ;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  ReportMemoryLeaksOnShutdown := True; // check for memory leaks
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Tutorial source code

The tutorial source code is included in the demo folder. It does not include a .proj file so you still need to add the Habari and Synapse source paths to the project options.

Map Message Tutorial

This tutorial provides a quick introduction to Habari Client for OpenMQ by walking you through the creation of a simple map message exchange application.

This tutorial takes less than 10 minutes to complete.

Setting up the project

To create a new project:

17. Start the Delphi IDE.
18. In the IDE, choose File > New > VCL Forms Application – Delphi
19. Choose Project > Options … to open the Project Options dialog
20. In the options tree on the left, select 'Delphi Compiler'
21. Add the source directory of Habari source, the Habari source\optional, and the Synapse source directory to the 'Search path'
22. Choose Ok to close the Project Options dialog
23. Save the project as HelloMapMessage

Now the project is created and saved.

You should see the main form in the GUI designer now.

Adding code to the project

To use the Habari Client for OpenMQ library, you need to add the required units to the source code.

24. Switch to Code view (F12)
25. Add the required units to the interface uses list:

```
uses
  BTConnectionFactory, BTJMSInterfaces, BTCommAdapterSynapse, BTJMSConnection,
  BTMessageTransformerXMLMapDocument, BTSerialIntf, BTTypes,
  // auto-generated unit references
  Windows, Messages, SysUtils, ...
```

26. Compile and save the project.
27. Switch to Design view (F12), go to the Tool palette (Ctrl+Alt+P) and add a TMemo and a TButton to the form.
28. Double click on the new button to jump to the Button Click handler
29. Add the following code to send the message:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
  MapMessage: IMapMessage;
  Key: PMString;
begin
  Factory := TBTConnectionFactory.Create;
  Connection := Factory.CreateConnection;
  SetTransformer(Connection, TBTMessageTransformerXMLMapDocument.Create(nil));
```

```
  Connection.Start;

  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMapMessage');
  Producer := Session.CreateProducer(Destination);

  MapMessage := Session.CreateMapMessage;
  MapMessage.SetString('DateTimeToStr(Now)', DateTimeToStr(Now));
  MapMessage.SetString('ParamStr(0)', ParamStr(0));

  Producer.Send(MapMessage);

  Memo1.Lines.Append('Sent:');
  for Key in MapMessage.GetMapNames do
  begin
    Memo1.Lines.Append(Key + '=' + MapMessage.GetString(Key));
  end;

  Connection.Close;
end;
```

30. Add a second button and double click on the new button to jump to the Button Click handler

31. Add the following code to receive and display the message:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Consumer: IMessageConsumer;
  MapMessage: IMapMessage;
  Key: PMString;
begin
  Factory := TBTConnectionFactory.Create;
  Connection := Factory.CreateConnection;
  SetTransformer(Connection, TBTMessageTransformerXMLMapDocument.Create(nil));
  Connection.Start;

  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMapMessage'
    + '?transformation=' + BTSerialIntf.TRANSFORMER_ID_MAP_XML);
  Consumer := Session.CreateConsumer(Destination);

  MapMessage := Consumer.Receive(1000) as IMapMessage;

  if Assigned(MapMessage) then
  begin
    Memo1.Lines.Append('Received:');
    for Key in MapMessage.GetMapNames do
    begin
      Memo1.Lines.Append(Key + '=' + MapMessage.GetString(Key));
    end;
  end;

  Connection.Close;
end;
```

32. Compile and save the project

Run the demo

- Launch the message broker
- Start the application
- Click on Button 1 to send the map message to the queue
- Click on Button 2 to receive the map message and display it

You can run two instances of the application at the same time, and also on different computers if the IP address of the message broker is used instead of localhost.

Map Message Conversion with Apache ActiveMQ

Note: if you send and receive map messages using the library, message brokers will receive them as simple STOMP text messages with a special header property "transformation" which is set to the value JMS_MAP_XML (or JMS_MAP_JSON if you use a JSON based map transformer class).

Most message brokers will not perform any special processing of these STOMP messages. A notable exception is Apache ActiveMQ: if the broker receives a STOMP message with the JMS_MAP_XML or JMS_MAP_JSON transformation header, it will convert the message internally to a 'native' JMS MapMessage. This allows Java clients to receive the message sent from the Delphi application as a MapMessage without the need to parse a XML body.

Habari Client map message transformers only support string properties.

Tutorial source code

The tutorial source code is included in the demo folder. It does not include a .proj file so you still need to add the Habari and Synapse source paths to the project options.

Connections and Sessions

Step-by-Step Example

Overview

This example will send a single message to a destination queue (ExampleQueue).[2]

Add required units

Three units are required for this example

- a communication adapter unit (e. g. BTCommAdapterIndy)
- a connection factory unit (BTConnectionFactory)
- the unit containing the interface declarations (BTJMSInterfaces)

The SysUtils unit is necessary for the exception handling.

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  BTCommAdapterIndy,
  BTConnectionFactory,
  BTJMSInterfaces,
  SysUtils;
...
```

Creating a new Connection

To create a new connection,

- declare a variable of type IConnection
- use an instance of TBTConnectionFactory to create connections

Since IConnection is an interface type, the connection instance will be destroyed automatically if there are no more references to it in the program.

```
var
  Factory: IConnectionFactory;
  Connection: IConnection;
```

2 Compatibility note: non-existing queues are created automatically by the broker - with the exception of HornetQ which requires them to be configured before usage

```
  ...
begin
  Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost');
  Connection := Factory.CreateConnection;
  ...
```

Connection URL Parameters

Connection URL parameters are documented in chapter "Connection URL parameters" and in chapter "Stomp 1.2".

Creating a Session

To create the communication session,

- declare a variable of type `ISession`
- use the helper method `CreateSession` of the connection, and specify the acknowledgment mode

Please check the API documentation for the different session types and acknowledgement modes.

Since `ISession` is an interface type, the session instance will be destroyed automatically if there are no more references to it in the program.

```
    Session := Connection.CreateSession(False, amAutoAcknowledge);
```

Using the Session

The Session variable is ready to use now. Destinations, producers and consumers will be covered in the next chapters.

```
    Destination := Session.CreateQueue('ExampleQueue');
    Producer := Session.CreateProducer(Destination);
    Producer.Send(Session.CreateTextMessage('This is a test message'));
```

Closing a Connection

Finally, the application closes the connection. The client will disconnect from the message broker. Closing a connection also implicitly closes all open sessions.

```
  finally
    Connection.Close;
  end;
end.
```

Note: Close will be called automatically if the connection is destroyed. But because unclosed connections use resources, Close should be called when the connection is no longer needed. When logging is enabled, the connection class will also log a message when a connection is destroyed without calling Close.

Session types overview

The table below shows the supported parameter combinations for the Connection.CreateSession method and their effect on the session transaction and acknowledgment features.

Parameters	**Client MUST acknowledge message receipt**[3]	**Transaction support for**		**STOMP Version**
		Send	Ack	
CreateSession(False, amAutoAcknowledge)	No	-	-	1.0
CreateSession(False, amClientAcknowledge)	Yes (cumulative effect)	-	-	1.0
CreateSession(False, amClientIndividual)	Yes	-	-	1.2
CreateSession(True, amAutoAcknowledge)	No	✓	-	1.0
CreateSession(True, amClientAcknowledge)	Yes (cumulative effect)	✓	✓①	1.0
CreateSession(True, amClientIndividual)	Yes	✓	✓①	1.2
CreateSession(True, amTransactional)	No	✓	-	1.0

① – ActiveMQ 5.9.0, Apollo 1.6, Open MQ 5.0.1 b08, RabbitMQ 3.2.3

Table 3: Session creation parameters

3 http://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE_ack_Header

Connection URL parameters

Message Receipts

The STOMP standard supports individual message receipts, which are sent back to the client for every message.[4]

The library supports this feature with an optional connection URL parameter, send.receipt.

Example:

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
send.receipt=true');
```

If the broker does not send a receipt within a time-out interval, the client library will raise an exception.

Note: for additional reliability, the client can use **transactional send** (see section "Transacted Sessions").

Subscription Receipts

The STOMP standard supports subscription receipts, which are sent back to the client for every subscription command.

The library supports this feature with an optional connection URL parameter, subscribe.receipt.

Example:

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
subscribe.receipt=true');
```

If the broker does not send a receipt within a time-out interval, the client library will raise an exception.

Disconnect Receipts

The STOMP specification recommends to request a broker confirmation for the DISCONNECT frame, to do a graceful shutdown, where the client is assured that all previous frames have been received by the server.[5]

The library supports this feature with an optional connection URL parameter, disconnect.receipt.

Example:

4 http://stomp.github.com/stomp-specification-1.1.html#Header_receipt
5 http://stomp.github.com/stomp-specification-1.1.html#DISCONNECT

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
disconnect.receipt=true');
```

Without this parameter, the client will disconnect the socket connection immediately after sending the DISCONNECT frame to the broker.

With disconnect.receipt=true, the client will send the DISCONNECT frame and then wait for up to 5000 milliseconds for the broker receipt frame. **If the broker does not answer, the client library will raise an exception.** The client application should treat its messages as undelivered.

Note:	for additional reliability, the client can use **transactional send** (see section "Transacted Sessions"), and message receipts (see section "Message Receipts").

TCP Keep-Alive (only supported with Indy)

The library supports TCP keep-alive with an optional connection URL parameter, tcp.keepalive.

The parameter takes two arguments. On Linux, the first argument is the initial delay before the first keep-alive, the second argument specifies the interval (both values are in milliseconds). On the Windows platform, the values of these arguments are ignored and the operating system uses default values for initial delay[6] and interval[7], which can be modified in the registry.

Example:

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
tcp.keepalive=1000,1000');
```

Note	TCP keep-alive is currenctly only supported by the Indy communication adapter
Important	To enable TCP keep-alive, the project must be compiled with the conditional symbol HABARI_TCP_KEEPALIVE

Heart-beating Support

STOMP 1.1 introduced heart-beating, its configuration is covered in the chapter Stomp 1.2

6 http://technet.microsoft.com/en-us/library/cc957549.aspx
7 http://technet.microsoft.com/en-us/library/cc957548.aspx

Transacted Sessions

A session may be specified as transacted. Each transacted session supports a single series of transactions.

Each transaction groups a set of message sends into an atomic unit of work.

A transaction is completed using either its session's Commit method or its session's Rollback method. The completion of a session's current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

Create a transacted session

To create a transacted session, set the parameter of CreateSession to amTransactional as shown in the code example:

```
Session := Connection.CreateSession(amTransactional);
```

or (using the older API version)

```
Session := Connection.CreateSession(True, amTransactional);
```

This code will automatically start a new transaction for this session.

Send messages

Now send messages using the transacted session.

```
Destination := Session.CreateQueue('testqueue');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('This is a test message'));
```

Committing a transaction

If your client code has successfully sent its messages, the transaction must be committed to make the messages visible on the destination.

```
// send messages ...

finally
  // commit all messages
  Session.Commit;
end;
```

Note: commiting a transaction automatically starts a new transaction

Rolling back a transaction

If your client code runs wants to undo the sending of its messages, the transaction may be rolled back, and the messages will not become visible on the destination.

```
// send messages ...

except
  ...
  // error!
  Session.Rollback;
  ...
end;
```

Note: rolling back a transaction automatically starts a new transaction. A transacted session will be rolled back automatically if the connection is closed.

Transacted message acknowledgement

Some library versions (see table "Communication Adapters" on page 13) support transactions also for the acknowledgement of received messages.

When a transaction is rolled back or the connection is closed without a commit, messages which have been acknowledged after the transaction start will return to unacknowledged state.

Code example:

```
// receive in a transacted session
Session := Connection.CreateSession(True, amClientAcknowledge);
Queue := Session.CreateQueue(GetQueueName);
Consumer := Session.CreateConsumer(Queue);
Msg := Consumer.Receive(1000);

// process the message
...

// acknowledge the message
Msg.Acknowledge;

...

// in case of errors, roll back all acknowledgements
Session.Rollback;
```

This is an experimental feature. It requires usage of the STOMP 1.2 communication protocol.

Failover Support

The Failover transport layers reconnect logic on top of the Stomp transport.[8]

The Failover configuration syntax allows you to specify any number of composite URIs. The Failover transport randomly chooses one of the composite URI and attempts to establish a connection to it. If it does not succeed, a new connection is established to one of the other URIs in the list.

Example for a failover URI:

```
failover:(stomp://primary:61613,stomp://secondary:61613)
```

Failover Transport Options

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
backOffMultiplier	2.0	The exponent used in the exponential backoff attempts
maxReconnectAttempts	-1	-1 is default and means retry forever, 0 means don't retry (only try connection once but no retry) If set to > 0, then this is the maximum number of reconnect attempts before an error is sent back to the client
randomize	true	use a random algorithm to choose the the URI to use for reconnect from the list provided

Table 4: Failover Transport Options

Example URI:

```
failover:(stomp://localhost:61616,stomp://remotehost:61616)?
initialReconnectDelay=100&maxReconnectAttempts=10
```

Example code:

8 http://activemq.apache.org/failover-transport-reference.html

```
Factory := TBTConnectionFactory.Create('failover:
(stomp://primary:61616,stomp://localhost:61613)?maxReconnectAttempts=3&randomize=false') do
try
  Conn := Factory.CreateConnection;
  Conn.Start;
  ...
  Conn.Stop;
finally
  Conn.Close;
end;
```

Pooled Connection Factory

A basic implementation of a connection pool class is included in the `libraries\objectpool` folder. With this class, multi-threaded applications can limit resource usage by reusing instances of IConnection.

To create a connection pool, the new class `TBTPoolableConnectionFactory` has been introduced. This class implements the `IPoolableInterfaceFactory` interface. It inherits from `TBTConnectionFactory` which makes it a drop-in replacement with the same methods to create and configure a classic, non-poolable connection factory.

Threads which have a reference to a `TBTConnectionPool` object then can use the methods `BorrowConnection` and `ReturnConnection` to acquire and release connections:

- **BorrowConnection** – this method returns a started IConnection instance
- **ReturnConnection** – this method returns the instance back to the pool

```
// acquire a connection
Conn := Pool.BorrowConnection;
try
  // ... use the connection
finally
  // return to pool
  Pool.ReturnConnection(Conn);
end;
```

The connection pool behaviour when the pool size is exhausted can be controlled by a constructor parameter with two different values, `eaFail` and `eaGrow`. The first value, `eaFail`, will raise an exception to notify the caller of the exhausted pool, so the caller needs to retry (or give up) when the pool has no free connection. The second value, `eaGrow`, will simply create another connection when no free connection is available.

Connections returned by `BorrowConnection` will be in started mode, and remain in this mode when they are returned to the pool.

Note: heart-beating is not supported, because connections in the pool will not automatically send heartbeats to the message broker.

A demo program is included in the `libraries/objectpool` folder.

Experimental Code This code is still in an experimental stage, and might change or be removed in later versions

Destinations

Introduction

The API supports two models:[9]

1. point-to-point or queuing model
2. publish and subscribe model

In the point-to-point or queuing model, a producer posts messages to a particular queue and a consumer reads messages from the queue. Here, the producer knows the destination of the message and posts the message directly to the consumer's queue. It is characterized by following:

- Only one consumer will get the message
- The producer does not have to be running at the time the receiver consumes the message, nor does the receiver need to be running at the time the message is sent
- Every message successfully processed is acknowledged by the receiver

The publish/subscribe model supports publishing messages to a particular message topic. Zero or more subscribers may register interest in receiving messages on a particular message topic. In this model, neither the publisher nor the subscriber know about each other. A good metaphor for it is anonymous bulletin board. The following are characteristics of this model:

- Multiple consumers can get the message
- There is a timing dependency between publishers and subscribers. The publisher has to create a subscription in order for clients to be able to subscribe. The subscriber has to remain continuously active to receive messages, unless it has established a durable subscription. In that case, messages published while the subscriber is not connected will be redistributed whenever it reconnects.

Create a new Destination

Queues

A queue can be created using the CreateQueue method of the Session. Example:

```
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
```

9 Java Message Service. (2007, November 21). In Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Java_Message_Service

The queue can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example)

Topics

A topic can be created using the CreateTopic method of the Session. Example:

```
Destination := Session.CreateTopic('bar');
Consumer := Session.CreateConsumer(Destination);
```

The topic can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example).

Producer and Consumer

Message Producer

A client uses a MessageProducer object to send messages to a destination. A MessageProducer object is created by passing a Destination object to a message-producer creation method supplied by a session.

Example:

```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('Test message'));
```

A client can specify a default delivery mode, priority, and time to live for messages sent by a message producer. It can also specify the delivery mode, priority, and time to live for an individual message.

Broker-specific note

Though the AMQP protocol supports the concept of priority, **RabbitMQ** does not yet implement that feature.[10]

Message Consumer

A client uses a MessageConsumer object to receive messages from a destination. A MessageConsumer object is created by passing a Destination object to a message-consumer creation method supplied by a session.

Example:

```
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
```

A message consumer can be created with a **message selector**. A message selector allows the client to restrict the messages delivered to the message consumer to those that match the selector.

10 http://dougbarth.github.com/2011/07/01/approximating-priority-with-rabbitmq.html

A client may either synchronously receive a message consumer's messages or have the consumer asynchronously deliver them as they arrive.

Synchronous

For synchronous receipt, a client can request the next message from a message consumer using one of its receive methods. There are several variations of receive that allow a client to poll or wait for the next message.

Asynchronous

For asynchronous delivery, a client can register a MessageListener object with a message consumer. As messages arrive at the message consumer, it delivers them by calling the MessageListener's OnMessage method.

Synchronous Receive

A MessageConsumer offers a Receive method which can be used to consume exactly one message at a time.

Example:

```
while I < EXPECTED do
begin
  TextMessage := Consumer.Receive(1000) as ITextMessage;
  if Assigned(TextMessage) then
  begin
    Inc(I);
    TextMessage.Acknowledge;
    L.Info(Format('%d %s', [I, TextMessage.Text]));
  end;
end;
```

Receive and ReceiveNoWait

There are three different methods for synchronous receive:

Receive	The Receive method with no arguments will block (wait until a message is available).
Receive(TimeOut)	The Receive method with a timeout parameter will wait for the given time in milliseconds. If no message arrived, it will return nil.
ReceiveNoWait	The ReceiveNowait method will return immediately. If no message arrived, it will return nil.

Should I use synchronous receive or IMessageListener?

Compared with *synchronous* receive, using a MessageListener for *asynchronous* delivery is less flexible. It is driven by an internal thread which does not allow further customization for specific use cases or advanced error handling.

In most cases it is recommended to move the message handling to a application-specific thread class, which creates the connection, reads messages using synchronous methods of the MessageConsumer object, and automatically reconnects if any errors occur.

Durable Subscriptions

Description

If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable TopicSubscriber.

The message broker retains a record of this durable subscription and insures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.[11]

The combination of the clientId and durable subscriber name uniquely identifies the durable topic subscription.

After you restart your program and re-subscribe, the broker will know which messages you need that were published while you were away.

Creation

The Session interface contains the CreateDurableSubscriber method which creates a durable subscriber to the specified topic.

A durable subscriber MessageConsumer is created with a unique clientID and durable subscriber name.

Only **one** thread can be actively consuming from a given logical topic subscriber.

Examples

Included unit tests for durable topics:

- TDurableSubTests.TestDurableSubWorksInNewSession

11 http://download.oracle.com/javaee/5/api/javax/jms/TopicSession.html

Temporary Queues

Introduction

> "Temporary destinations (temporary queues or temporary topics) are proposed as a lightweight alternative in a scalable system architecture that could be used as unique destinations for replies. Such destinations have a scope limited to the connection that created it, and are removed on the server side as soon as the connection is closed." ("Designing Messaging Applications with Temporary Queues", by Thakur Thribhuvan [12])

Library Support

Temporary destinations are supported by

- ActiveMQ
- Apollo
- OpenMQ
- RabbitMQ

Resource Management

The session should be closed as soon as processing is completed so that TemporaryQueues will be deleted on the server side.

12 http://onjava.com/pub/a/onjava/2007/04/10/designing-messaging-applications-with-temporary-queues.html

Message Options

JMS Standard Properties

The Apache ActiveMQ message broker supports some JMS standard properties in the STOMP adapter. These properties are based on the JMS specification of the Message interface.[13]

Note: If your application makes use of these properties, your application depends on a broker-specific feature which is not available on other in the STOMP adapter of other message brokers

JMS properties for outgoing messages

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.
JMSPriority	The message priority level.
JMSReplyTo	The Destination object to which a reply to this message should be sent.

JMS properties for incoming messages

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.
JMSPriority	The message priority level.
JMSTimestamp	The timestamp the broker added to the message.

13 http://download.oracle.com/javaee/5/api/javax/jms/Message.html

JMSMessageId	The message ID which is set by the provider.
JMSReplyTo	The Destination object to which a reply to this message should be sent.

Reserved Names

The following names are reserved Stomp header properties and can not be used as names for user defined properties:

- activemq.* (everything starting with activemq is a reserved name)
- login
- passcode
- transaction
- session
- message
- destination
- id
- ack
- selector
- type
- content-length
- correlation-id
- expires
- persistent
- priority
- reply-to
- message-id
- timestamp
- transformation
- client-id
- redelivered

The client library detects overwriting of Stomp defined message properties. It will raise an Exception if the application tries to send a message with a reserved property name.

JMS Selectors

Selectors are a way of attaching a filter to a subscription to perform content based routing. For more documentation on the detail of selectors see the reference on javax.jmx.Message[14].

Supported message brokers

Message selectors are supported by

- Habari Client for ActiveMQ
- Habari Client for Apollo
- Habari Client for HornetQ
- Habari Client for OpenMQ

Code example:

```
Consumer := Session.CreateConsumer(Destination, 'type=''car'' and color=''blue''');
```

All supported brokers allow supports string type properties and operations in selectors. Two versions support also allow integer properties and operations in selectors:

- Habari Client for ActiveMQ – see special note[15]
- Habari Client for Apollo

Code example:

```
Consumer := Session.CreateConsumer(Destination, 'type=''car'' and color=''blue'' and weight >
2500');
```

14 http://docs.oracle.com/javaee/5/api/javax/jms/Message.html
15 http://activemq.apache.org/selectors.html

Object Messages

Introduction

Object Serialization

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time.[16] In messaging applications, object serialization is required to transfer objects between clients, but also to store objects on the broker if they are declared persistent.

Message Transformers

Transformation	Message Type	Library	Unit
XML	**ObjectMessage**	**OmniXML**	BTMessageTransformerXMLOmni
XML	**ObjectMessage**	**NativeXml**	BTMessageTransformerXMLNative
XML	**MapMessage**	**OmniXML**	BTMessageTransformerXMLMapOmni
XML	**MapMessage**	**NativeXml**	BTMessageTransformerXMLMapNative
XML	**MapMessage**	**IDocument**	BTMessageTransformerXMLMapDocument
JSON	**ObjectMessage**	**SuperObject**	BTMessageTransformerJSONSuperObject
JSON	**MapMessage**	**SuperObject**	BTMessageTransformerJSONMapSuperObject

Table 5: Message Transformer Implementations

Memory Management

Outgoing Objects

The message transformer will not free objects which have been sent. To release the memory, the application has to explicitly free them when they are no longer used.

Incoming Objects

The message transformer will create an object instance when a object message has been received. To avoid memory leaks, the application must free this instance when it is no longer in use.

16 http://java.sun.com/developer/technicalArticles/Programming/serialization/

Assign a Message Transformer

To insert a object decoder / encoder in the message processing chain, create a message transformer instance and assign it to the connection's MessageTransformer property.

The constructor of message transformers for object exchange takes one argument, which is the class of the serialized object. In this example, SamplePojo is the class.

```
Connection: IConnection;
...

with (Connection as IMessageTransfomerSupport) do
begin
  MessageTransformer := TBTMessageTransformerXMLOmni.Create(SamplePojo);
end;

...
Connection.Start;
```

You can also use the helper procedure SetTransformer in unit BTJMSConnection:

```
Connection: IConnection;
...

SetTransformer(Connection, TBTMessageTransformerXMLOmni.Create(SamplePojo));

...
Connection.Start;
```

Create and Send an ObjectMessage

1. create a IObjectMessage instance using ISession#CreateObjectMessage
2. send the object message to the broker using IMessageProducer#Send

```
ObjectMessage := Session.CreateObjectMessage(Instance);
Producer.Send(ObjectMessage);
```

Complete Example using NativeXml

From ObjectExchangeTests.pas.

Send:

```
procedure TObExTestCase.TestXMLNative;
var
  ObjectMessage: IObjectMessage;
  Obj: SamplePojo;
begin
  // send
```

```
  Connection := TBTJMSConnection.MakeConnection;
  try
    SetTransformer(Connection, TBTMessageTransformerXMLNative.Create(SamplePojo));
    Connection.Start;
    Session := Connection.CreateSession(False, amAutoAcknowledge);
    Destination := Session.CreateQueue('TOOL.OBJECT.XML');
    Producer := Session.CreateProducer(Destination);
    Obj := SamplePojo.Create;
    try
      Obj.messageText := 'test';
      Obj.messageNo := 0;
      ObjectMessage := Session.CreateObjectMessage(Obj);
      ObjectMessage.SetStringProperty(SH_TRANSFORMATION + '-custom',
        TRANSFORMER_ID_OBJECT_XML); // required for "Delphi Only" object exchange
      Producer.Send(ObjectMessage);
    finally
      Obj.Free;
    end;
  finally
    Connection.Close;
  end;
```

Receive:

```
  Connection := TBTJMSConnection.MakeConnection;
  try
    SetTransformer(Connection, TBTMessageTransformerXMLNative.Create(SamplePojo));
    Connection.Start;
    Session := Connection.CreateSession(False, amClientAcknowledge);
    Destination := Session.CreateQueue('TOOL.OBJECT.XML');
    Consumer := Session.CreateConsumer(Destination);
    ObjectMessage := Consumer.Receive(1000) as IObjectMessage;
    if Assigned(ObjectMessage) then
    begin
      ObjectMessage.Acknowledge;
      Obj := ObjectMessage.GetObject as SamplePojo;
      try
        CheckEquals('test', Obj.messageText);
        CheckEquals(0, Obj.messageNo);
      finally
        Obj.Free;
      end;
    end;
  finally
    Connection.Close;
  end;
end;
```

Stomp 1.2

Connection Configuration

A connection string can use additional URL parameters to configure Stomp version 1.1 and 1.2

All parameters can be omitted to use the default value.

Switch	Description	Default
connect.accept-version[17]	Supported Stomp versions in ascending order	No default
connect.host[18]	**The name of a virtual host that the client wishes to connect to.** It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.	Server URI
connect.heart-beat[19]	Heart beat (outgoing, incoming)	"0,0"

Connection Factory Code Example:

```
Factory := TBTConnectionFactory.Create(
  'stomp://localhost:61613?connect.accept-version=1.2&connect.heart-beat=1000,0');
```

This example creates a connection factory with these connection settings

host: localhost

17 http://stomp.github.com//stomp-specification-1.2.html#protocol_negotiation
18 http://stomp.github.com//stomp-specification-1.2.html#CONNECT_or_STOMP_Frame
19 http://stomp.github.com//stomp-specification-1.2.html#Heart-beating

port: 61613

accept-version: 1.2

heart-beat: 1000,0

- virtual host is localhost
- the client requests Stomp 1.2 protocol
- client heart beat interval is 1000 milliseconds, no server heart beat signals

Specification

For details see the Stomp specification pages:

http://stomp.github.com//stomp-specification-1.1.html

http://stomp.github.com//stomp-specification-1.2.html

Sending heart-beat signals

A client can use the **SendHeartbeat** method of the connection object to send a heart-beat byte (newline 0x0A).

SendHeartbeat is a method of the IHeartbeat interface, which is declared in the BTSessionIntf unit. A cast of the IConnection object is required to access this method.

Code example:

```
(Connection as IHeartbeat).SendHeartbeat;
```

Notes:

- the client application code is responsible for sending a heartbeat message within the maximum interval which was specified in the connect parameter – the Habari Client library does not send heart-beats automatically
- client messages which are sent after the heart-beat interval expires may be lost

Checking for incoming heartbeats

The Habari client library stores a time-stamp of the last incoming data. If the time which elapsed since this time-stamp is greater than two times the heart-bet interval, calling **CheckHeartbeat** will raise an exception of type EBTStompServerHeartbeatMissing.

Code example:

```
(Connection as IHeartbeat).CheckHeartbeat;
```

Notes:

- the method raises an exception if the connection does not use server-side heart-beating
- the method only checks the time elapsed since the last heart-beat, it does not try to read any data from the connection

Reading server-side heartbeats

If the client never needs to consume any messages, but still needs to check for server-side heartbeats, it can use the **ReceiveHeartbeat** method of the connection object.

This method takes one argument, TimeOut.

The function returns True if it found at least one heart-beat signal on the connection.

Calling ReceiveHeartbeat is only useful for applications which never call Receive, to check if the server is still healthy, and to consume the pending heart-beat signals from the connection.

If the client reads messages (using Consumer.Receive), calling ReceiveHeartbeat is not required.

Example Applications

Directory	Description
common	Shared units (see below)
common-chat	Simple chat client
common-consumertool	Receive messages from broker
common-consumertool-fpc	Free Pascal version of ConsumerTool
common-delphigui	Sends and receives messages
common-heartbeat-server	Uses server-side heart-beating to check the connection / server health (not supported on all message brokers[20])
common-performance	Multi-threaded performance test application
common-producertool	Send messages to broker
common-producertool-fpc	Free Pascal version of ProducerTool
common-producertool-ssl	Send messages to broker with SSL connection
common-reconnect	Send messages and reconnect on connection failure
common-rpc	Use temporary queues to implement request/response style communication (not supported on all message brokers[21])
common-tests	DUnit tests (requires DUnit version 9.4)
common-tests-fpc	FPCUnit tests (requires Lazarus 1.2 or newer)
common-throughput	Continuously produces and consumes messages to monitor the average message throughput over time
common-throughput-fpc	Free Pascal version of ThroughputTest

Table 6: Example Applications (in alphabetic order)

20 Requires STOMP 1.1 or 1.2; not supported by OpenMQ
21 Not available with the HornetQ message broker

Broker-independent example projects

The demo folder contains projects for example applications and unit tests. Their folder name begins with common- to indicate their code is shared between all Habari Client libraries.

Broker-specific preparation

Important note: message brokers which do not automatically create destinations (such as HornetQ) need to be configured before running the demos.

Shared units for demo projects

The directory `demo/common` contains shared units:

- connection configuration form (ConnCfgFrm.pas/dfm)
- command line parameter support class (CommandLineSupport.pas)
- console appender for the Log4D logging framework

SSL communication adapter units

The directory source/optional contains example implementations of Indy and Synapse adapter classes with OpenSSL support. Please note that these are basic implementations and not supported in the free basic support plan.

ConsumerTool

The ConsumerTool demo may be used to receive messages from a queue or topic. This example application is configurable by command line parameters, all are optional.

Parameter	Default Value	Description
AckMode	CLIENT_ACKNOWLEDGE	Acknowledgment mode, possible values are: CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or SESSION_TRANSACTED
ClientId		Client Id for durable subscriber
ConsumerName	Habari	name of the message consumer - for durable subscriber
Durable	false	true: use a durable subscriber
MaximumMessages	10	expected number of messages
Password		Password
PauseBeforeShutDown	false	true: wait for key press
ReceiveTimeOut	0	0: asynchronous receive, > 0: consume messages while they continue to be delivered within the given time out
SleepTime	0	time to sleep after asynchronous receive
Subject	TOOL.DEFAULT	queue or topic name
Topic	false	true: topic false: queue
Transacted	false	true: transacted session
URL	localhost	server url
User		user name
Verbose	true	verbose output

Table 7: ConsumerTool Command Line Options

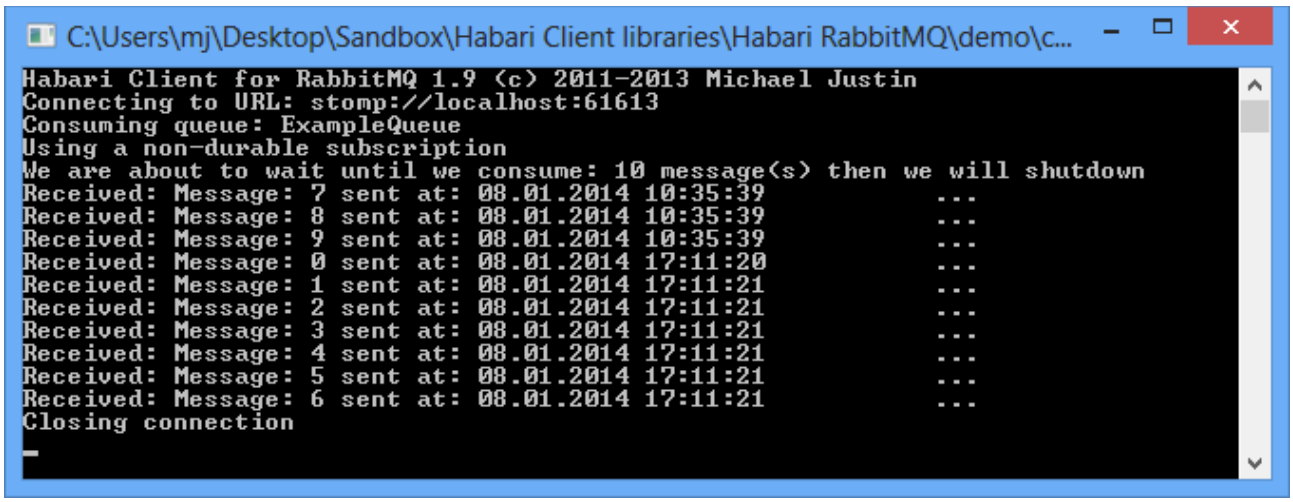

C:\Users\mj\Desktop\Sandbox\Habari Client libraries\Habari RabbitMQ\demo\c...
Habari Client for RabbitMQ 1.9 (c) 2011-2013 Michael Justin
Connecting to URL: stomp://localhost:61613
Consuming queue: ExampleQueue
Using a non-durable subscription
We are about to wait until we consume: 10 message(s) then we will shutdown
Received: Message: 7 sent at: 08.01.2014 10:35:39 ...
Received: Message: 8 sent at: 08.01.2014 10:35:39 ...
Received: Message: 9 sent at: 08.01.2014 10:35:39 ...
Received: Message: 0 sent at: 08.01.2014 17:11:20 ...
Received: Message: 1 sent at: 08.01.2014 17:11:21 ...
Received: Message: 2 sent at: 08.01.2014 17:11:21 ...
Received: Message: 3 sent at: 08.01.2014 17:11:21 ...
Received: Message: 4 sent at: 08.01.2014 17:11:21 ...
Received: Message: 5 sent at: 08.01.2014 17:11:21 ...
Received: Message: 6 sent at: 08.01.2014 17:11:21 ...
Closing connection

Illustration 2: ConsumerTool demo application

Examples

Receive 1000 messages from local broker

```
ConsumerTool --MaximumMessages=1000
```

Receive 10 messages from local broker and wait for any key

```
ConsumerTool --PauseBeforeShutDown
```

Use a transacted session to receive 10,000 messages from local broker

```
ConsumerTool --MaximumMessages=10000 --Transacted --AckMode=SESSION_TRANSACTED
```

ProducerTool

The ProducerTool demo can be used to send messages to the broker. It is configurable by command line parameters, all are optional.

Parameter	Default	Description
MessageCount	10	Number of messages
MessageSize	255	Length of a message in bytes
Persistent	false	Delivery mode 'persistent'
SleepTime	0	Pause between messages in milliseconds
Subject	TOOL.DEFAULT	Destination name
TimeToLive	0	Message expiration time
Topic	false	Destination is a topic
Transacted	false	Use a transaction
URL	localhost	Message broker URL
Verbose	true	Verbose output
User		User name
Password		Password

Table 8: ProducerTool Command Line Options

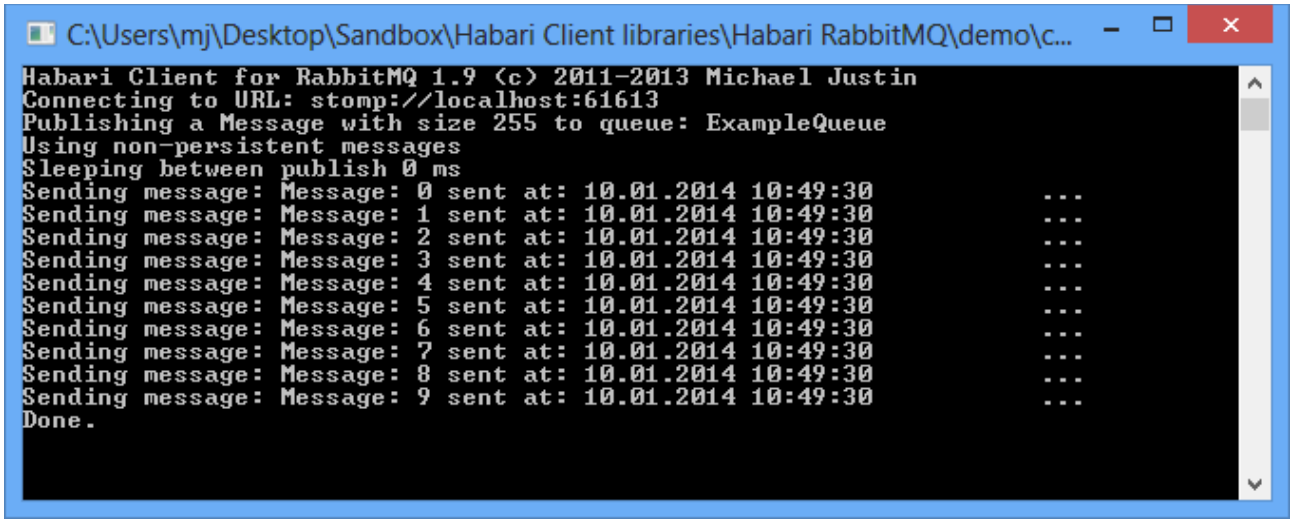

C:\Users\mj\Desktop\Sandbox\Habari Client libraries\Habari RabbitMQ\demo\c...
Habari Client for RabbitMQ 1.9 (c) 2011-2013 Michael Justin
Connecting to URL: stomp://localhost:61613
Publishing a Message with size 255 to queue: ExampleQueue
Using non-persistent messages
Sleeping between publish 0 ms
Sending message: Message: 0 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 1 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 2 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 3 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 4 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 5 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 6 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 7 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 8 sent at: 10.01.2014 10:49:30 ...
Sending message: Message: 9 sent at: 10.01.2014 10:49:30 ...
Done.

Illustration 3: ProducerTool demo application

Examples

Send 10,000 messages to the queue `TOOL.DEFAULT` on the local broker

```
ProducerTool --MessageCount 10000
```

```
```

Send 10 messages to the topic ExampleTopic on the local broker

```
ProducerTool --Topic --Subject=ExampleTopic
```

Performance Test

The performance test application provides a GUI for multi-threaded sending and receiving of messages.

- A broker configuration dialog can be invoked by clicking the URL field
- The communication library (Indy or Synapse) can be selected
- Number and length of messages and thread number can be adjusted using the sliders

For every thread a message queue with the name ExampleQueue.<n> will be used.

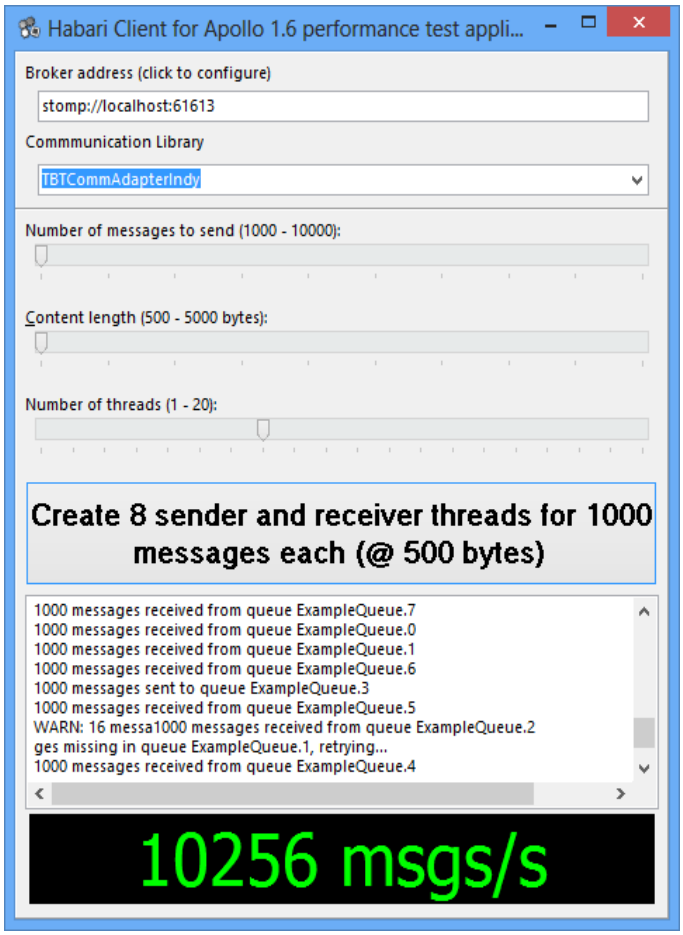

Habari Client for Apollo 1.6 performance test appli...
Broker address (click to configure)
stomp://localhost:61613
Commmunication Library
TBTCommAdapterIndy
Number of messages to send (1000 - 10000):
Content length (500 - 5000 bytes):
Number of threads (1 - 20):
Create 8 sender and receiver threads for 1000 messages each (@ 500 bytes)
1000 messages received from queue ExampleQueue.7
1000 messages received from queue ExampleQueue.0
1000 messages received from queue ExampleQueue.1
1000 messages received from queue ExampleQueue.6
1000 messages sent to queue ExampleQueue.3
1000 messages received from queue ExampleQueue.5
WARN: 16 messa1000 messages received from queue ExampleQueue.2
ges missing in queue ExampleQueue.1, retrying...
1000 messages received from queue ExampleQueue.4
10256 msgs/s

Illustration 4: Performance Test Application

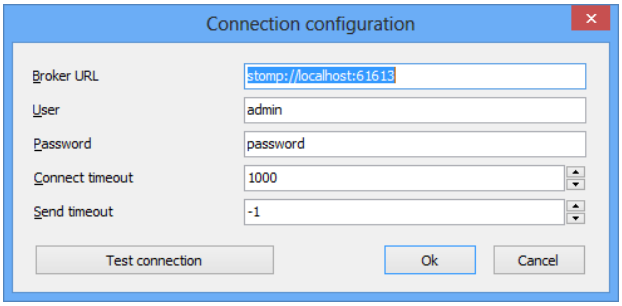


Illustration 5: Click in the URL field to open the configuration dialog

Throughput Test Tool

This example application is configurable by command line parameters, all are optional.

Parameter	Default Value	Description
Password	(broker-specific)	Password
Subject	ExampleTopic	Topic name
URL	(broker-specific)	Connection URL
User	(broker-specific)	User name

Table 9: Throughput Test Tool Command Line Options

Examples

Use remote broker 'mybroker' and specify user and password

```
tptest --url=stomp://mybroker --user=test1 --password=secret
```

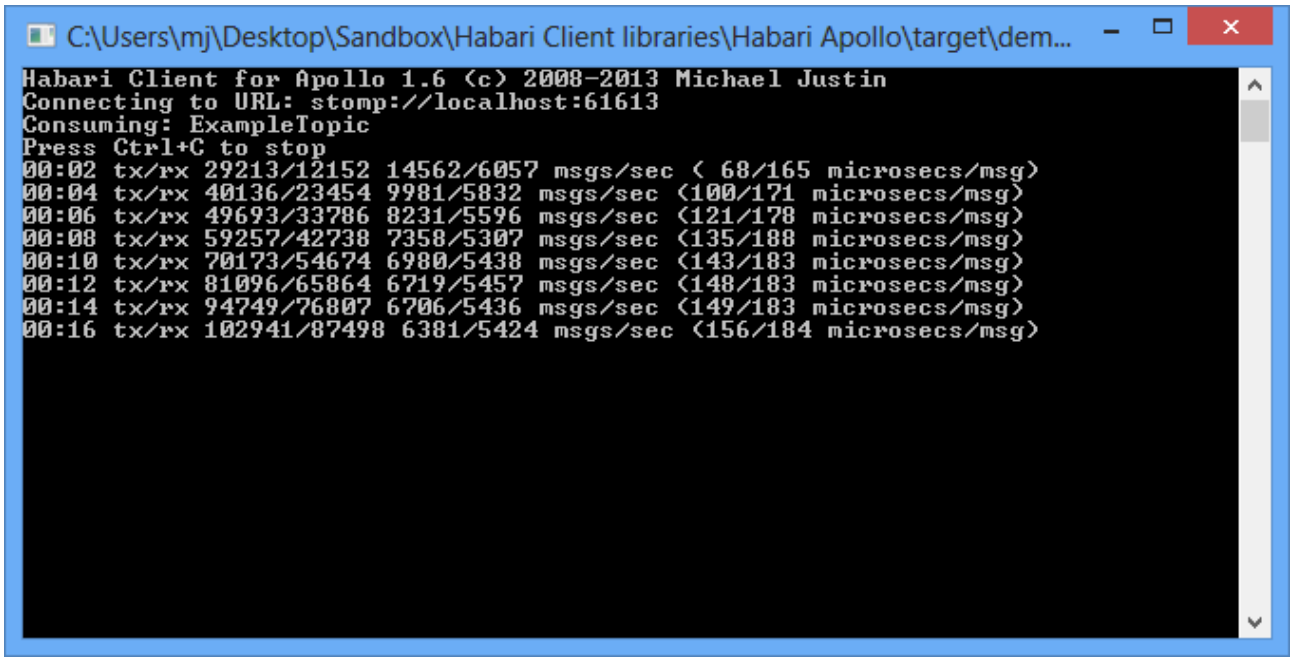

C:\Users\mj\Desktop\Sandbox\Habari Client libraries\Habari Apollo\target\dem...
Habari Client for Apollo 1.6 (c) 2008-2013 Michael Justin
Connecting to URL: stomp://localhost:61613
Consuming: ExampleTopic
Press Ctrl+C to stop
00:02 tx/rx 29213/12152 14562/6057 msgs/sec (68/165 microsecs/msg)
00:04 tx/rx 40136/23454 9981/5832 msgs/sec (100/171 microsecs/msg)
00:06 tx/rx 49693/33786 8231/5596 msgs/sec (121/178 microsecs/msg)
00:08 tx/rx 59257/42738 7358/5307 msgs/sec (135/188 microsecs/msg)
00:10 tx/rx 70173/54674 6980/5438 msgs/sec (143/183 microsecs/msg)
00:12 tx/rx 81096/65864 6719/5457 msgs/sec (148/183 microsecs/msg)
00:14 tx/rx 94749/76807 6706/5436 msgs/sec (149/183 microsecs/msg)
00:16 tx/rx 102941/87498 6381/5424 msgs/sec (156/184 microsecs/msg)

Illustration 6: Throughput test tool output

Logging with Log4D

Introduction

Habari Client libraries include the free open source library Log4D as an optional dependency. Some patches of the library are included to resolve issues with Free Pascal and newer versions of Delphi.

IDE configuration

In order to compile with Log4D support, add the conditional symbol `HABARI_LOGGING` to the project options:

- in Delphi, choose Project | Options... | Delphi Compiler > Conditional defines and add HABARI_LOGGING
- in Lazarus, choose Project | Project Options … | Compiler Options > Other and add -dHABARI_LOGGING in the Custom options field

Minimal example

To log to the "Event log" window in the Delphi IDE, just add the Log4D unit to the uses list and initialize Log4D with this line:

```
TlogBasicConfigurator.Configure; // creates ODS (OutputDebugString) logger with level
'error'
```

and set the log level to fatal, error, warn, info, debug or trace with another line:

```
TLogLogger.GetRootLogger.Level := Trace; // or any other log level
```

Log4D configuration file

Log4D can be configured in the source code at application start-up or using a configuration file. A configuration allows to reconfigure logging without recompiling. Note that the application only reads the configuration file at start-up, so it needs to be restarted after modifications.

In your project start-up code, configure the Log4D framework with the path to a Log4D configuration file.

Example:

```
TLogPropertyConfigurator.Configure(ExtractFilePath(ParamStr(0)) +
'config/log4d.props');
```

Example file "log4d.props"

To log to the IDE output window using ODS (OutputDebugString), the minimal Log4D configuration shown below can be used:

```
# Set internal debugging
log4d.configDebug=false

# Global logging level
log4d.threshold=info

# Set logger factory
log4d.loggerFactory=TLogDefaultLoggerFactory

# Set root level
log4d.rootLogger=info,ODS

# Create root appender
log4d.appender.ODS=TLogODSAppender
log4d.appender.ODS.layout=TLogPatternLayout
log4d.appender.ODS.layout.pattern=%p | %c - %m%n
log4d.appender.ODS.layout.dateFormat=nn:ss.zzz
```

Logging to ODS and to the console window

The configuration below also logs messages to the console window. This example configuration uses the unit LogConsoleAppender which is included in the demo/commons folder. It is not part of the Log4D framework and only an example implementation provided for your convenience, but without support.

Important To use this configuration, the main project file must contain the LogConsoleAppender unit

```
# Set internal debugging
log4d.configDebug=false

# Global logging level
log4d.threshold=info

# Set logger factory
log4d.loggerFactory=TLogDefaultLoggerFactory

# Set root level
log4d.rootLogger=info,ODS,Con1

# Create root appender
log4d.appender.ODS=TLogODSAppender
log4d.appender.ODS.layout=TLogPatternLayout
log4d.appender.ODS.layout.pattern=%p | %c - %m%n
log4d.appender.ODS.layout.dateFormat=nn:ss.zzz
```

```
# Create a console appender
log4d.appender.Con1=TLogConsoleAppender
log4d.appender.Con1.append=true
log4d.appender.Con1.errorHandler=TLogOnlyOnceErrorHandler
log4d.appender.Con1.layout=TLogPatternLayout
log4d.appender.Con1.layout.pattern= %-5p| %m%n
log4d.appender.Con1.layout.dateFormat=hh:nn:ss.zzz
```

Log level

In the configuration shown above, the log level is set to “info” so it will only process log messages with level info and higher (warnings and errors):

```
# Global logging level
log4d.threshold=info
```

For debug purposes it can be helpful to show detailed messages. You can enable trace logs by setting the global logging level or by setting the logging level for an individual log appender.

```
# Global logging level
log4d.threshold=trace
```

Conditional Symbols

HABARI_LOGGING

This conditional symbol enables logging support. Logging support requires the open source logging framework Log4D

Log4D is available on Sourceforge at http://log4d.sourceforge.net/

HABARI_RAW_TRACE

Enables detailed logging of Stomp message frames.

If this symbol is defined, a compiler warning will be emitted:

```
Compiled with HABARI_RAW_TRACE
```

Useful Units

BTStreamHelper

This unit contains the procedure LoadBytesFromStream which can be used to read a file into a BytesMessage.

Example:

```
// create the message
Msg := Session.CreateBytesMessage;

// open a file
FS := TFileStream.Create('filename.dat', fmOpenRead);

try
  // read the file bytes into the message
  LoadBytesFromStream(Msg, FS);

  Size := Length(Msg.Content);

  // display message content size
  WriteLn(IntToStr(Size) + ' Bytes');

finally
  // release the file stream
  FS.Free;
end;
```

BTJavaPlatform

This unit contains some helper functions for Java dates. Java dates are Int64 values based on the Unix date.

```
function JavaDateToTimeStamp(const JavaDate: Int64): TDateTime;
```

```
function TimeStampToJavaDate(const TimeStamp: TDateTime): Int64;
```

Library Limitations

Session

Transacted Sessions

Transactional sending

The library supports transactional **sending** of messages with ActiveMQ, Apollo, HornetQ, OpenMQ and RabbitMQ.

Transactional acknowledging

Transactional **acknowledging** of incoming messages has been implemented and tested with Apache ActiveMQ 5.9.0, Apollo 1.7 and RabbitMQ 3.3.0.

HornetQ and OpenMQ do not support transactional acknowledging of incoming messages.

MessageConsumer

How do I implement synchronous receive from multiple destinations?

The library does not support synchronous receive from more than one destination over a **single** connection.

To receive messages synchronously (using Receive and ReceiveNoWait) from two or more destinations, create one connection per destination.

Background: all pending messages in a connection are serialized in one TCP stream, so reading only the messages which come from one of the destinations would require 'skipping' all messages for other destinations.

Asynchronous receive using a MessageListener is not affected by this limitation.

Message Properties

Only string data type supported by Stomp

The STOMP protocol uses string type key/value lists for the representation of message properties. Regardless of the method used to set message properties, all message properties will be interpreted as Java Strings by the Message Broker.

As a side effect, the expressions in a Selector are limited to operations which are valid for strings.

Timestamp properties are converted to a Unix time stamp value, which is the internal representation in Java. But still, these values can not be used with date type expressions.

Broker-specific exceptions

Apache ActiveMQ 5.6 introduced support for numeric expressions in JMS selectors[22].

Multi Threading

GUI application detection using System.IsConsole

GUI applications require thread synchronization with the main VCL thread.

The library internally uses the `System.IsConsole` flag to detect if it is a console or GUI application and uses `synchronize()` if `System.IsConsole` is `False`.

However calling synchronize in a non-GUI application (for example a ISAPI module) will block the program execution.

If the library is used in a non-GUI application and the library seems to hang, make sure that `System.IsConsole` is set to `True`.

Setting `{$APPTYPE CONSOLE}` in the project source file is the easiest way to fix this.

Other limitations

A session supports transactions and it is difficult to implement transactions that are multi-threaded; a session should not be used concurrently by multiple threads.

22 https://issues.apache.org/jira/browse/AMQ-1609

ISAPI modules

ISAPI modules can use the library, but they must set `{$APPTYPE CONSOLE}`. The Habari Client library internally uses the `System.IsConsole` flag to detect if it is a console or GUI application and uses `synchronize()` if `System.IsConsole` is `False`.

Online Resources

Third-Party Libraries

Internet Direct (Indy)

Project home	http://www.indyproject.org/
Documentation	http://www.indyproject.org/Sockets/Docs/index.aspx
Snapshot	http://indy.fulgan.com/ZIP/
Subversion	https://svn.atozed.com:444/svn/Indy10/trunk
Subversion docs	http://www.indyproject.org/Sockets/Download/svn.aspx

Synapse

Project home	http://www.synapse.ararat.cz
Subversion	https://synalist.svn.sourceforge.net/svnroot/synalist/trunk/

Log4D

Project home	http://sourceforge.net/projects/log4d/
Subversion	svn://svn.code.sf.net/p/log4d/code/trunk

NativeXml

Project home	http://www.simdesign.nl/xml.html
Google home	http://simdesign.googlecode.com/
Subversion	http://simdesign.googlecode.com/svn/trunk/

OmniXML

Project home	http://code.google.com/p/omnixml/
Subversion	http://omnixml.googlecode.com/svn/trunk/

SuperObject

Project home	https://code.google.com/p/superobject/
GIT	https://code.google.com/p/superobject/

Specifications

Stomp – Simple (or Streaming) Text Oriented Messaging Protocol[23]

Stomp home	http://stomp.github.io/index.html
Stomp 1.2	http://stomp.github.io/stomp-specification-1.2.html
Stomp 1.1	http://stomp.github.io/stomp-specification-1.1.html
Stomp 1.0	http://stomp.github.io/stomp-specification-1.0.html

Broker-specific documentation

ActiveMQ	http://activemq.apache.org/stomp.html
Apollo	http://activemq.apache.org/apollo/documentation/stomp-manual.html
RabbitMQ	https://www.rabbitmq.com/stomp.html

23 http://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol

Online Articles

Title	Broker
Firebird Database Events and Message-oriented Middleware[24]	All
Introducing connection pooling for Habari Client libraries[25]	All
Discover ActiveMQ brokers with Delphi XE4 and Indy 10.6[26]	ActiveMQ
Official RabbitMQ Management REST API Documentation[27]	RabbitMQ
How to use the RabbitMQ Web-Stomp Plugin[28]	RabbitMQ
RPC with Delphi client and Java server using RabbitMQ[29]	RabbitMQ
Delphi Integration with the GlassFish v3 Application Server[30]	OpenMQ

Online Videos

Title	Broker

24 http://mikejustin.wordpress.com/2012/11/06/firebird-database-events-and-message-oriented-middleware/
25 http://mikejustin.wordpress.com/2013/06/06/introducing-connection-pooling-for-habari-client-libraries/
26 http://mikejustin.wordpress.com/2013/07/07/discover-activemq-brokers-with-delphi-xe4-and-indy-10-6/
27 http://mikejustin.wordpress.com/2012/10/26/official-rabbitmq-management-rest-api-documentation/
28 http://mikejustin.wordpress.com/2013/11/27/how-to-use-the-rabbitmq-web-stomp-plugin-with-delphi-and-free-pascal/
29 http://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/
30 https://mikejustin.fogbugz.com/default.asp?W11

Introduction to Messaging With Apache ActiveMQ[31]	ActiveMQ
GlassFish Message Queue - High Availability Clusters[32]	OpenMQ

31 http://vimeo.com/12654513
32 http://www.youtube.com/watch?v=RHUJBsy3udU

Support

Bug reports

A public bug tracking system is available at https://mikejustin.fogbugz.com/ - no login is required. Please select the product from the list of projects so we know which library version is affected by the problem.

Support inquiries

Please send support inquiries to support@habarisoft.com

To allow fast processing of your inquiry, please provide a detailed problem description, including configuration and environment, or code examples which help to reproduce the problem.

Advanced support

Advanced and experimental features such as (for example) SSL, third party libraries, Free Pascal, Linux, non-Unicode Delphi versions and message broker configuration are not covered by the basic support scheme.

Index

Reference

Asynchronous receive....62
BorrowConnection....30
BTCommAdapterIndy....22
BTJMSConnection....43
CheckHeartbeat....47
Configuration file....58
Connect.accept-version....45
Connect.heart-beat....45
Connect.host....45
Connection....22
Connection URL....23
ConnectionFactory....22
ConnectTimeout....13
ConsumerTool....50
CreateDurableSubscriber....37
CreateObjectMessage....43
Destination....32
Disconnect Receipts....25
Disconnect.receipt....25
Failover Support....29
HABARI_LOGGING....57, 60
HABARI_RAW_TRACE....60
IConnection....22
IHeartbeat....46
IMessageProducer....43
Internet Direct (Indy)....**11**
IPoolableInterfaceFactory....30
ISAPI....64
IsConsole....63
ISession....43
JMS Selector....41
JMSCorrelationID....39
JMSDeliveryMode....39
JMSExpiration....39
JMSMessageId....40
JMSPriority....39
JMSReplyTo....39
JMSTimestamp....39
Limitations....13, 62
Log4D....57, 60
Log4D configuration....57
Log4d.props....58
LogConsoleAppender....58
Logging....57
Message Consumer....34
Message Producer....34
Message properties....63
Message Receipts....25
MessageListener....35, 62
MessageTransformer....43
Multi Threading....63
Multiple destinations....62
NativeXml....42
Object Message....42
ODS....58
OmniXML....42
OnMessage....35
OpenSSL....49
OutputDebugString....58
Point-to-point....32
ProducerTool....52
Programming Model....15
Publish and subscribe....32
Queue....32
Receive....35
ReceiveHeartbeat....47
ReceiveNoWait....35
ReturnConnection....30
SamplePojo....43
Selector....63
Send.receipt....25
SendHeartbeat....46
Session....23
SetTransformer....43
Stomp 1.2....45
Subscribe.receipt....25
SuperObject....42
Synapse....**11,** 13
Synchronous receive....62
TBTConnectionFactory....22, 30
TBTConnectionPool....30
TBTPoolableConnectionFactory....30
TCP....62
Tcp.keepalive....26
Throughput Test Tool....56
Topic....33
TopicSubscriber....37
Transacted Sessions....27, 62
Transactions....63
Virtual host....45
$APPTYPE....63

Illustrations

Illustration 1: Programming Model.....15
Illustration 2: ConsumerTool demo application.....50
Illustration 3: ProducerTool demo application.....52
Illustration 4: Performance Test Application.....54
Illustration 5: Click in the URL field to open the configuration dialog.....55
Illustration 6: Throughput test tool output.....56

Tables

Table 1: Communication Adapters.....13
Table 2: Communication Adapters with SSL Support.....14
Table 3: Session creation parameters.....24
Table 4: Failover Transport Options.....29
Table 5: Message Transformer Implementations.....42
Table 6: Example Applications (in alphabetic order).....48
Table 7: ConsumerTool Command Line Options.....50
Table 8: ProducerTool Command Line Options.....52
Table 9: Throughput Test Tool Command Line Options.....56